Recall: Namespaces for communication over IP

- Hostname
  - www.eecs.berkeley.edu
- IP address
  - 128.32.244.172 (ipv6?)
- Port Number
  - 0-1023 are "well known" or "system" ports
    - Superuser privileges to bind to one
  - 1024 - 49151 are "registered" ports (registry)
    - Assigned by IANA for specific services
  - 49152-65535 (2^15+2^14 to 2^16−1) are "dynamic" or "private"
    - Automatically allocated as "ephemeral Ports"

Recall: Use of Sockets in TCP

- Socket: an abstraction of a network I/O queue
  - Embodies one side of a communication channel
    - Same interface regardless of location of other end
    - Could be local machine (called “UNIX socket”) or remote machine (called “network socket”)
  - First introduced in 4.2 BSD UNIX: big innovation at time
    - Now most operating systems provide some notion of socket
- Using Sockets for Client-Server (C/C++ interface):
  - On server: set up “server-socket”
    - Create socket, Bind to protocol (TCP), local address, port
    - Call listen(): tells server socket to accept incoming requests
    - Perform multiple accept() calls on socket to accept incoming connection request
    - Each successful accept() returns a new socket for a new connection; can pass this off to handler thread
  - On client:
    - Create socket, Bind to protocol (TCP), remote address, port
    - Perform connect() on socket to make connection
    - If connect() successful, have socket connected to server

Recall: Socket Setup over TCP/IP

- Server Socket: Listens for new connections
  - Produces new sockets for each unique connection
- Things to remember:
  - Connection involves 5 values:
    [ Client Addr, Client Port, Server Addr, Server Port, Protocol ]
  - Often, Client Port “randomly” assigned
    - Done by OS during client socket setup
  - Server Port often “well known”
    - 80 (web), 443 (secure web), 25 (sendmail), etc
    - Well-known ports from 0–1023
Example: Server Protection and Parallelism

Client

Create Client Socket

Connect it to server (host:port)

write request

read response

Close Client Socket

Server

Create Server Socket

Bind it to an Address (host:port)

Listen for Connection

Accept connection

write response

Close Connection Socket

Close Server Socket

Recall: Server Protocol (v3)

while (1) {
    listen(lstnsockfd, MAXQUEUE);
    consockfd = accept(lstnsockfd, (struct sockaddr *) &cli_addr,
                      &clilen);
    cpid = fork();              /* new process for connection */
    if (cpid > 0) {             /* parent process */
        close(consockfd);
    } else if (cpid == 0) {      /* child process */
        close(lstnsockfd);        /* let go of listen socket */
        server(consockfd);
        close(consockfd);
        exit(EXIT_SUCCESS);         /* exit child normally */
    }
}
close(lstnsockfd);

Server Address - itself

- Simple form
- Internet Protocol
- accepting any connections on the specified port
- In “network byte ordering”

Client: getting the server address

struct hostent *buildServerAddr(struct sockaddr_in *serv_addr,
                              char *hostname, int portno) {
    struct hostent *server;
    /* Get host entry associated with a hostname or IP address */
    server = gethostbyname(hostname);
    if (server == NULL) {
        fprintf(stderr,"ERROR, no such host\n");
        exit(1);
    }
    /* Construct an address for remote server */
    memset((char *) &serv_addr, 0, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = INADDR_ANY;
    serv_addr.sin_port = htons(portno);
    return server;
}
BIG OS Concepts so far

- Processes
- Address Space
- Protection
- Dual Mode
- Interrupt handlers (including syscall and trap)
- File System
  - Integrates processes, users, cwd, protection
- Key Layers: OS Lib, Syscall, Subsystem, Driver
  - User handler on OS descriptors
- Process control
  - fork, wait, signal, exec
- Communication through sockets
- Client-Server Protocol

Recall: Traditional UNIX Process

- Process: Operating system abstraction to represent what is needed to run a single program
  - Often called a “HeavyWeight Process”
  - No concurrency in a “HeavyWeight Process”
- Two parts:
  - Sequential program execution stream
    - Code executed as a sequential stream of execution (i.e., thread)
    - Includes State of CPU registers
  - Protected resources:
    - Main memory state (contents of Address Space)
    - I/O state (i.e. file descriptors)

How do we Multiplex Processes?

- The current state of process held in a process control block (PCB):
  - This is a “snapshot” of the execution and protection environment
  - Only one PCB active at a time
- Give out CPU time to different processes (Scheduling):
  - Only one process “running” at a time
  - Give more time to important processes
- Give pieces of resources to different processes (Protection):
  - Controlled access to non-CPU resources
  - Example mechanisms:
    - Memory Mapping: Give each process their own address space
    - Kernel/User duality: Arbitrary multiplexing of I/O through system calls
CPU Switch From Process to Process

- This is also called a “context switch”
- Code executed in kernel above is overhead
  - Overhead sets minimum practical switching time
  - Less overhead with SMT/hyperthreading, but... contention for resources instead

Lifecycle of a Process

- As a process executes, it changes state:
  - new: The process is being created
  - ready: The process is waiting to run
  - running: Instructions are being executed
  - waiting: Process waiting for some event to occur
  - terminated: The process has finished execution

Process Scheduling

- PCBs move from queue to queue as they change state
  - Decisions about which order to remove from queues are Scheduling decisions
  - Many algorithms possible (few weeks from now)

Ready Queue And Various I/O Device Queues

- Thread not running ⇒ TCB is in some scheduler queue
  - Separate queue for each device/signal/condition
  - Each queue can have a different scheduler policy
Administrivia

- Group signups: 4 members/group
  - Sign up with the autograder
  - Form asks which section you attend
- Need to get to know your Tas
  - Consider moving out of really big sections!
- Finding info on your own is a good idea!
  - Learn your tools, like “man”
  - Can even type “man xxx” into google!
    » Example: “man ls”

Modern Process with Threads

- Thread: *a sequential execution stream within process* (Sometimes called a “Lightweight process”)
  - Process still contains a single Address Space
  - No protection between threads
- Multithreading: *a single program made up of a number of different concurrent activities*
  - Sometimes called multitasking, as in Ada …
- Why separate the concept of a thread from that of a process?
  - Discuss the “thread” part of a process (concurrency)
  - Separate from the “address space” (protection)
    - Heavyweight Process ≡ Process with one thread

Single and Multithreaded Processes

- Threads encapsulate concurrency: “Active” component
- Address spaces encapsulate protection: “Passive” part
  - Keeps buggy program from trashing the system
- Why have multiple threads per address space?

Thread State

- State shared by all threads in process/addr space
  - Content of memory (global variables, heap)
  - I/O state (file descriptors, network connections, etc)
- State “private” to each thread
  - Kept in TCB = Thread Control Block
  - CPU registers (including, program counter)
  - Execution stack - what is this?
- Execution Stack
  - Parameters, temporary variables
  - Return PCs are kept while called procedures are executing
### Execution Stack Example

- **A** (int tmp) {
  - if (tmp<2) B();
  - printf(tmp);
}
- **B** () {
  - **C** () {
    - **A** (2);
  }  
- **A** (1);

- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

```
A(int tmp) {
  if (tmp<2) B();
  printf(tmp);
}
B() {
  C();
}
C() {
  A(2);
}
A(1);
```

### Motivational Example for Threads

- Imagine the following C program:
  ```c
  main() {
    ComputePI("pi.txt");
    PrintClassList("clist.text");
  }
  ```

- What is the behavior here?
  - Program would never print out class list
  - Why? ComputePI would never finish

### Use of Threads

- Version of program with Threads (loose syntax):
  ```c
  main() {
    ThreadFork(ComputePI("pi.txt"));
    ThreadFork(PrintClassList("clist.text"));
  }
  ```

- What does “ThreadFork()” do?
  - Start independent thread running given procedure
- What is the behavior here?
  - Now, you would actually see the class list
  - This should behave as if there are two separate CPUs

### Memory Footprint: Two-Threads

- If we stopped this program and examined it with a debugger, we would see
  - Two sets of CPU registers
  - Two sets of Stacks

- Questions:
  - How do we position stacks relative to each other?
  - What maximum size should we choose for the stacks?
  - What happens if threads violate this?
  - How might you catch violations?
Actual Thread Operations

- **thread_fork(func, args)**
  - Create a new thread to run func(args)
  - Pintos: thread/create

- **thread_yield()**
  - Relinquish processor voluntarily
  - Pintos: thread_yield

- **thread_join(thread)**
  - In parent, wait for forked thread to exit, then return

- **thread_exit**
  - Quit thread and clean up, wake up joiner if any
  - Pintos: thread_exit

- pThreads: POSIX standard for thread programming

Dispatch Loop

- Conceptually, the dispatching loop of the operating system looks as follows:

  ```
  Loop {
    RunThread();
    ChooseNextThread();
    SaveStateOfCPU(curTCB);
    LoadStateOfCPU(newTCB);
  }
  ```

- This is an infinite loop
  - One could argue that this is all that the OS does
  - Should we ever exit this loop???
    - When would that be?

Running a thread

Consider first portion: RunThread()

- How do I run a thread?
  - Load its state (registers, PC, stack pointer) into CPU
  - Load environment (virtual memory space, etc)
  - Jump to the PC

- How does the dispatcher get control back?
  - Internal events: thread returns control voluntarily
  - External events: thread gets preempted

Internal Events

- **Blocking on I/O**
  - The act of requesting I/O implicitly yields the CPU

- **Waiting on a “signal” from other thread**
  - Thread asks to wait and thus yields the CPU

- **Thread executes a yield()**
  - Thread volunteers to give up CPU

  ```
  computePI() {
    while(TRUE) {
      ComputeNextDigit();
      yield();
    }
  }
  ```
Stack for Yielding Thread

- How do we run a new thread?
  
  ```c
  run_new_thread() {
    newThread = PickNewThread();
    switch(curThread, newThread);
    ThreadHouseKeeping(); /* Do any cleanup */
  }
  ```

- How does dispatcher switch to a new thread?
  - Save anything next thread may trash: PC, regs, stack
  - Maintain isolation for each thread

What do the stacks look like?

- Consider the following code blocks:
  ```c
  proc A() {
    B();
  }
  ```
  ```c
  proc B() {
    while(TRUE) {
      yield();
    }
  }
  ```

Suppose we have 2 threads:
- Threads S and T

Saving/Restoring state (often called “Context Switch”)

```c
Switch(tCur,tNew) {
  /* Unload old thread */
  TCB[tCur].regs.r7 = CPU.r7;
  ...
  TCB[tCur].regs.r0 = CPU.r0;
  TCB[tCur].regs.sp = CPU.sp;
  TCB[tCur].regs.retpc = CPU.retpc; /*return addr*/

  /* Load and execute new thread */
  CPU.r7 = TCB[tNew].regs.r7;
  ...
  CPU.r0 = TCB[tNew].regs.r0;
  CPU.sp = TCB[tNew].regs.sp;
  CPU.retpc = TCB[tNew].regs.retpc;
  return; /* Return to CPU.retpc */
}
```

Switch Details (continued)

- What if you make a mistake in implementing switch?
  - Suppose you forget to save/restore register 4
  - Get intermittent failures depending on when context switch occurred and whether new thread uses register 4
  - System will give wrong result without warning

- Can you devise an exhaustive test to test switch code?
  - No! Too many combinations and inter-leavings

  **Cautionary tail:**
  - For speed, Topaz kernel saved one instruction in switch()
  - Carefully documented!
    - Only works As long as kernel size < 1MB
  - What happened?
    - Time passed, People forgot
    - Later, they added features to kernel (no one removes features)
    - Very weird behavior started happening

  Moral of story: Design for simplicity
### Some Numbers

- Frequency of performing context switches: 10-100ms
- Context switch time in Linux: 3-4 μsecs (Current Intel i7 & E5).
  - Thread switching faster than process switching (100 ns).
  - But switching across cores about 2x more expensive than within-core switching.
- Context switch time increases sharply with the size of the working set*, and can increase 100x or more.

* The working set is the subset of memory used by the process in a time window.

**Moral:** Context switching depends mostly on cache limits and the process or thread’s hunger for memory.

### What happens when thread blocks on I/O?

- What happens when a thread requests a block of data from the file system?
  - User code invokes a system call
  - Read operation is initiated
  - Run new thread/switch

- Thread communication similar
  - Wait for Signal/Join
  - Networking

![Diagram showing stack growth and execution flow]

### External Events

- What happens if thread never does any I/O, never waits, and never yields control?
  - Could the ComputePI program grab all resources and never release the processor?
    - What if it didn’t print to console?
  - Must find way that dispatcher can regain control!

**Answer:** Utilize External Events

- Interrupts: signals from hardware or software that stop the running code and jump to kernel
- Timer: like an alarm clock that goes off every some many milliseconds

- If we make sure that external events occur frequently enough, can ensure dispatcher runs

### Thread Abstraction

<table>
<thead>
<tr>
<th>Programmer Abstraction</th>
<th>Physical Reality</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads</td>
<td>1 2 3 4 5</td>
</tr>
<tr>
<td>Processes</td>
<td>1 2 3 4 5</td>
</tr>
</tbody>
</table>

- Infinite number of processors
- Threads execute with variable speed
  - Programs must be designed to work with any schedule
### Programmer vs. Processor View

<table>
<thead>
<tr>
<th>Programmer's View</th>
<th>Possible Execution #1</th>
<th>Possible Execution #2</th>
<th>Possible Execution #3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$x = x + 1$;</td>
<td>$x = x + 1$;</td>
<td>$x = x + 1$;</td>
<td>$x = x + 1$;</td>
</tr>
<tr>
<td>$y = y + x$;</td>
<td>$y = y + x$;</td>
<td>$y = y + x$;</td>
<td>$y = y + x$;</td>
</tr>
<tr>
<td>$z = x + 5y$;</td>
<td>$z = x + 5y$;</td>
<td>thread is suspended</td>
<td>$z = x + 5y$;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>other thread(s) run</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>thread is resumed</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>other thread(s) run</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>thread is resumed</td>
<td></td>
</tr>
</tbody>
</table>

### Possible Executions

- **a)** One execution
- **b)** Another execution

### Thread Lifecycle

- **Init**
  - Thread Creation: e.g., thread_create()
- **Ready**
  - Scheduler Suspends Thread: e.g., thread_yield()
- **Running**
  - Thread Yields to Scheduler: e.g., thread_exit()
  - Thread Waits for Event: e.g., thread_join()
- **Waiting**
  - Event Occurs: e.g., other thread calls thread_join()
- **Finished**

### Shared vs. Per-Thread State

<table>
<thead>
<tr>
<th>Shared State</th>
<th>Per-Thread State</th>
<th>Per-Thread State</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Heap</strong></td>
<td>Thread Control Block (TCB)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Stack Information</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Saved Registers</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Thread Metadata</td>
<td></td>
</tr>
<tr>
<td><strong>Global Variables</strong></td>
<td>Thread Control Block (TCB)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Stack Information</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Saved Registers</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Thread Metadata</td>
<td></td>
</tr>
<tr>
<td><strong>Code</strong></td>
<td>Stack</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Stack</td>
<td></td>
</tr>
</tbody>
</table>
Per Thread State (Kernel Supported Threads)

- Each Thread has a **Thread Control Block (TCB)**
  - Execution State: CPU registers, program counter (PC), pointer to stack (SP)
  - Scheduling info: state, priority, CPU time
  - Various Pointers (for implementing scheduling queues)
  - Pointer to enclosing process (PCB) – user threads
  - Etc (add stuff as you find a need)

- OS Keeps track of TCBs in “kernel memory”
  - In Array, or Linked List, or …

Multithreaded Processes

- PCB points to multiple TCBs:
  - Switching threads within a block is a simple thread switch
  - Switching threads across blocks requires changes to memory and I/O address tables.

Examples multithreaded programs

- Embedded systems
  - Elevators, Planes, Medical systems, Wristwatches
  - Single Program, concurrent operations

- Most modern OS kernels
  - Internally concurrent because have to deal with concurrent requests by multiple users
  - But no protection needed within kernel

- Database Servers
  - Access to shared data by many concurrent users
  - Also background utility processing must be done

Example multithreaded programs (con’t)

- Network Servers
  - Concurrent requests from network
  - Again, single program, multiple concurrent operations
  - File server, Web server, and airline reservation systems

- Parallel Programming (More than one physical CPU)
  - Split program into multiple threads for parallelism
  - This is called Multiprocessing

- Some multiprocessors are actually uniprogrammed:
  - Multiple threads in one address space but one program at a time
A typical use case

Client Browser
- process for each tab
- thread to render page
- GET in separate thread
- multiple outstanding GETs
- as they complete, render portion

Web Server
- fork process for each client connection
- thread to get request and issue response
- fork threads to read data, access DB, etc
- join and respond

Some Actual Numbers

- Many process are multi-threaded, so thread context switches may be either within-process or across-processes.

Kernel Use Cases

- Thread for each user process
- Thread for sequence of steps in processing I/O
- Threads for device drivers
- ...

Putting it together: Process

(Unix) Process

A(int tmp) {
    if (tmp < 2)
        B();
        printf(tmp);
    }
B() {
    C();
}
C() {
    A(2);
}
A(1);

Resources

Sequential stream of instructions

CPU state (PC, SP, registers...)

I/O State (e.g., file, socket contexts)

Memory

Stack

Stored in OS
Putting it together: Processes

- **Switch overhead**: high
  - CPU state: low
  - Memory/IO state: high
- **Process creation**: high
- **Protection**
  - CPU: yes
  - Memory/IO: yes
- **Sharing overhead**: high
  (involves at least a context switch)

Switch overhead: high
- CPU state: low
- Memory/IO state: high
- Process creation: high
- Protection
  - CPU: yes
  - Memory/IO: yes
- Sharing overhead: high
  (involves at least a context switch)

Switch overhead: low (only CPU state)
- Thread creation: low
- Protection
  - CPU: yes
  - Memory/IO: No
- Sharing overhead: low (thread switch overhead low)

Kernel versus User-Mode threads

- We have been talking about Kernel threads
  - Native threads supported directly by the kernel
  - Every thread can run or block independently
  - One process may have several threads waiting on different things
- Downside of kernel threads: a bit expensive
  - Need to make a crossing into kernel mode to schedule
- Even lighter weight option: User Threads
  - User program provides scheduler and thread package
  - May have several user threads per kernel thread
  - User threads may be scheduled non-preemptively relative to each other (only switch on yield())
  - Cheap
- Downside of user threads:
  - When one thread blocks on I/O, all threads block
  - Kernel cannot adjust scheduling among all threads
  - Option: Scheduler Activations
    » Have kernel inform user level when thread blocks...

Simple One-to-One Threading Model

- Many-to-One
- Many-to-Many
Threads in a Process

- Threads are useful at user-level
  - Parallelism, hide I/O latency, interactivity
- Option A (early Java): user-level library, within a single-threaded process
  - Library does thread context switch
  - Kernel time slices between processes, e.g., on system call I/O
- Option B (SunOS, Unix variants): green Threads
  - User-level library does thread multiplexing
- Option C (Windows): scheduler activations
  - Kernel allocates processors to user-level library
  - Thread library implements context switch
  - System call I/O that blocks triggers upcall
- Option D (Linux, MacOS, Windows): use kernel threads
  - System calls for thread fork, join, exit (and lock, unlock, …)
  - Kernel does context switching
  - Simple, but a lot of transitions between user and kernel mode

Putting it together: Multi-Cores

- Switch overhead: low (only CPU state)
- Thread creation: low
- Protection
  - CPU: yes
  - Memory/IO: No
- Sharing overhead: low (thread switch overhead low, may not need to switch at all!)

Putting it together: Hyper-Threading

- Switch overhead between hardware-threads: very-low (done in hardware)
- Contention for ALUs/FPUs may hurt performance

Multiprocessing vs Multiprogramming

- Remember Definitions:
  - Multiprocessing = Multiple CPUs
  - Multiprogramming = Multiple Jobs or Processes
  - Multithreading = Multiple threads per Process
- What does it mean to run two threads “concurrently”?
  - Scheduler is free to run threads in any order and interleaving: FIFO, Random, …
  - Dispatcher can choose to run each thread to completion or time-slice in big chunks or small chunks

Multiprocessing

Multiprogramming
Single and Multithreaded Processes

Supporting 1T and MT Processes

Correctness for systems with concurrent threads

- If dispatcher can schedule threads in any way, programs must work under all circumstances
  - Can you test for this?
  - How can you know if your program works?
- Independent Threads:
  - No state shared with other threads
  - Deterministic ⇒ Input state determines results
  - Reproducible ⇒ Can recreate Starting Conditions, I/O
  - Scheduling order doesn’t matter (if switch() works!!!)
- Cooperating Threads:
  - Shared State between multiple threads
  - Non-deterministic
  - Non-reproducible
  - Non-deterministic and Non-reproducible means that bugs can be intermittent
  - Sometimes called “Heisenbugs”
Interactions Complicate Debugging

- Is any program truly independent?
  - Every process shares the file system, OS resources, network, etc
  - Extreme example: buggy device driver causes thread A to crash "independent" thread B

- You probably don’t realize how much you depend on reproducibility:
  - Example: Evil C compiler
    » Modifies files behind your back by inserting errors into C program unless you insert debugging code
  - Example: Debugging statements can overrun stack

- Non-deterministic errors are really difficult to find
  - Example: Memory layout of kernel+user programs
    » depends on scheduling, which depends on timer/other things
    » Original UNIX had a bunch of non-deterministic errors
  - Example: Something which does interesting I/O
    » User typing of letters used to help generate secure keys

Why allow cooperating threads?

- People cooperate; computers help/enhance people’s lives, so computers must cooperate
  - By analogy, the non-reproducibility/non-determinism of people is a notable problem for "carefully laid plans"

- Advantage 1: Share resources
  - One computer, many users
  - One bank balance, many ATMs
    » What if ATMs were only updated at night?
  - Embedded systems (robot control: coordinate arm & hand)

- Advantage 2: Speedup
  - Overlap I/O and computation
    » Many different file systems do read-ahead
  - Multiprocessors - chop up program into parallel pieces

- Advantage 3: Modularity
  - More important than you might think
  - Chop large problem up into simpler pieces
    » To compile, for instance, gcc calls cpp | cc1 | cc2 | as | ld
    » Makes system easier to extend

High-level Example: Web Server

- Server must handle many requests

Non-cooperating version:

```c
serverLoop() {
    con = AcceptCon();
    ProcessFork(ServiceWebPage(),con);
}
```

What are some disadvantages of this technique?

Threaded Web Server

- Now, use a single process

Multithreaded (cooperating) version:

```c
serverLoop() {
    connection = AcceptCon();
    ThreadFork(ServiceWebPage(),connection);
}
```

- Looks almost the same, but has many advantages:
  - Can share file caches kept in memory, results of CGI scripts, other things
  - Threads are much cheaper to create than processes, so this has a lower per-request overhead

- Question: would a user-level (say one-to-many) thread package make sense here?
  - When one request blocks on disk, all block...

- What about Denial of Service attacks or digg / Slash-dot effects?
Thread Pools

- Problem with previous version: Unbounded Threads
  - When web-site becomes too popular – throughput sinks
- Instead, allocate a bounded “pool” of worker threads, representing the maximum level of multiprogramming

```c
master() {
    allocThreads(worker,queue);
    while(TRUE) {
        con=AcceptCon();
        if (con==null)
            sleepOn(queue);
        else
            ServiceWebPage(con);
        wakeUp(queue);
    }
}

worker(queue) {
    while(TRUE) {
        con=Dequeue(queue);
        if (con==null)
            sleepOn(queue);
        else
            ServiceWebPage(con);
    }
}
```

Classification

<table>
<thead>
<tr>
<th># threads</th>
<th># of addr-spaces</th>
</tr>
</thead>
<tbody>
<tr>
<td>One</td>
<td>One</td>
</tr>
<tr>
<td>Many</td>
<td>Many</td>
</tr>
</tbody>
</table>

- One or many address spaces
- One or many threads per address space
- Did Windows 95/98/ME have real memory protection?
  - No: Users could overwrite process tables/System DLLs

Summary

- Processes have two parts
  - Threads (Concurrency)
  - Address Spaces (Protection)
- Concurrency accomplished by multiplexing CPU Time:
  - Unloading current thread (PC, registers)
  - Loading new thread (PC, registers)
  - Such context switching may be voluntary (yield(), I/O operations) or involuntary (timer, other interrupts)
- Protection accomplished restricting access:
  - Memory mapping isolates processes from each other
  - Dual-mode for isolating I/O, other resources
- Various Textbooks talk about processes
  - When this concerns concurrency, really talking about thread portion of a process
  - When this concerns protection, talking about address space portion of a process